Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

sanitize

- Le projet sanitizers porté par Google contient plusieurs outils
- ASan : Address Sanitize permet l'injection dans l'exécutable de tests concernant les erreurs mémoires.
- Cela permet en particulier :
  - Utilisation de mémoire libérée (free)
  - Dépassement de tableau de pile, de tas ou global
  - Fuites mémoires
- Le surcoût temps et mémoire est très réduit.

sanitize

- L’activation de l’option se fait lors de la compilation et édition de lien en ajoutant l’option : -fsanitize=address
- L’ajout de l’option -g permet d’avoir des messages plus complets
- La variable d’environnement ASAN_OPTIONS permet d’agir sur l’activation ou la désactivations de certains tests.
- ASan est compatible avec gdb

Sanitize vs Valgrind

- ASan n’est pas compatible avec valgrind
- L’avantage de ASan est que le surcoût d’exécution est très faible et l’on peut donc l’activer par défaut quand on est en phase de développement.
- Valgrind détecte plus d’erreurs, comme l’utilisation de variables non initialisées.
- L’activation de ASan nécessite la recompilation complète du projet

clang-tidy

- clang-tidy est un analyseur statique de code
- C’est une plateforme qui permet l’intégration de nombreux tests.
- Il permet de nombreux tests sur les fichiers sources à la recherche de bug potentiels.
- Il permet également de vérifier le respect d’une convention de codage (nommage variable/fonction par ex.)

clang-tidy

```
#include <stdio.h>

int main(int argc, char **argv){
  int i;
  if (argc>3) i=0;
  printf("%d\n",i);
}
```

```
$ clang-tidy foo.c -- -I.
1 warning generated.
/tmp/pg120/foo.c:6:3: warning: 2nd function call argument is an uninitialized value [clang-analyzer-core.CallAndMessage]
  printf("%d\n",i);
  ^           ~
/tmp/pg120/foo.c:4:3: note: 'i' declared without an initial value
  int i;
  ^~~~~
/tmp/pg120/foo.c:5:7: note: Assuming 'argc' is <= 3
  if (argc>3) i=0;
      ^~~~~~
/tmp/pg120/foo.c:5:3: note: Taking false branch
  if (argc>3) i=0;
  ^
/tmp/pg120/foo.c:6:3: note: 2nd function call argument is an uninitialized value
  printf("%d\n",i);
  ^           ~
```

gdb/valgrind/ASan/clang-tidy

- En résumé, il est recommandé de tester votre code avec clang-tidy **avant** de le compiler.
- Par défaut, on peut compiler avec ASan son projet, cela permet d'avoir plus d'information en cas de bug
- L'utilisation de valgrind en systématique va dépendre de contrainte en performance et malheureusement vous ne pouvez compiler pour ASan et valgrind
- Gdb vous permettra de tracker un bug pour en trouver l'origine. Il permet également d'inspecter le déroulement d'un algorithme (plus efficace et moins intrusif que le printf).

Convention de codage

- Une convention de codage est un document qui liste les règles d'écriture de code source pour un projet / une entreprise :
  - nommage des fonctions, variables, macro…
  - nommage et organisation de fichiers
  - langue pour le code et les commentaires
  - formatage spécifique (boucles, tests…)
  - techniques de programmation
- Une convention est un document vivant qui doit être mis à jour si nécessaire.

convention de codage : exemple

- https://www.kernel.org/doc/Documentation/CodingStyle
  - indentation
  - taille max de lignes
  - positionnement des accolades et parenthèses
  - espaces
  - nommage : C is a Spartan language, and so should your naming be….
  - typedef
  - fonctions
  - sortie de fonctions
  - commentaires
  - … .

Convention de codage & clang-format

- L'outil *clang-format* permet de reformater automatiquement du code source selon une convention.
- Il contient déjà plusieurs conventions de codage : LLVM, Google, Chromium, Mozilla, WebKit
- Il est possible de définir sa convention via un fichier .clang-format
- Ne permet pas d'implémenter **toutes** les règles, on le combine avec clang-tidy pour avoir une vérification plus exhaustive.

Documentation

- La documentation est primordiale pour un projet sur le long terme
- Plusieurs niveaux de doc :
  - très haut niveau : présentation large, vue d'ensemble, éléments d'architecture
  - modules : aspects fonctionnels, périmètre
  - fonction : description des paramètres, valeurs de retours, spécifications
  - code : astuces mises en œuvre, point d'algorithmique non trivial, justification de choix.

Documentation

- Le maintient d'une documentation peut être un travail long et il arrive souvent qu'il y ai divergence entre la documentation et le code.
- Pour cela, on rapproche la documentation du code en l'incluant dans celui-ci
- Utilisation des commentaires et de générateurs de documentation
- Ne permet pas de faire toute la documentation (en particulier, la doc de haut niveau, manuel d'utilisation...).

doxygen

- doxygen permet de générer la documentation au format html/pdf/latex… à partir des commentaires dans le code source.
- doxygen peut également intégrer de la documentation au format Markdown
- Le résultat est une documentation séparée des sources mais synchronisée avec celles-ci.

doxygen

- doxygen utilise des commentaires suivant certaines règles d'écriture :

```
//! power function
/*! The pow() function returns the value of x raised to the power of y.
* \param x a real in double format
* \param y a real in double format
* \return x raised to power of y or NaN if wrong arguments
*/
double pow(double x, double y) ;
```

- doxygen gère d'autres formats (javadoc par exemple).

doxygen

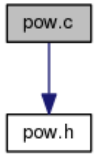
- A partir des commentaires structurés des sources, doxygen va générer une documentation dans différents formats :
  - Html
  - Latex
  - Page de manuel (man pages)
  - ...

My Project

pow.c File Reference

```
#include "pow.h"
```

Include dependency graph for pow.c:



Go to the source code of this file.

Functions

double **pow** (double x, double y)

Function Documentation

```
double pow ( double  x,
             double  y
           )
```

power function The **pow()** function returns the value of x raised to the power of y.

Parameters

x a real in double format

y a real in double format

Returns

x raised to power of y or NaN if wrong arguments

Definition at line **5** of file **pow.c**.

doxygen et README.md

- Il est souhaitable pour un projet d'avoir un fichier qui donne des informations générales de haut niveau :
  - auteurs
  - objectif de l'application
  - pré-requis
  - installation
  - utilisation
  - …
- Une technique consiste à donner ces éléments dans un fichier README.md à la racine du projet. Ce fichier pourra être intégré à la documentation par doxygen :

```
My great project          {#mainpage}
==============

About
--------
```

My Project

Main Page | Files

My Project
- My great project
- Files

My great project

Author

Julien Allali

About

example for PG106 course.

Commentaires : qq règles

- Les commentaires doivent être **utiles**
- Pour une fonction :
  - ce que fait la fonction (spécification)
  - éventuellement, comment elle le fait (algo, complexité, coût mémoire…)
  - **domaine de valeur des paramètres**
  - cas d'erreurs
- Les commentaires dans le code doivent servir à suivre la logique de celui-ci, par ex :
  - // set default value into the matrix
  - …
  - // fill the matrix according to the formula : M[i][j]=min(M[i-1][j],M[i][j-1])
  - …
  - // backtrace to compute the alignment
  - …

Commentaires : qq règles

- Pour les modules, penser à ajouter une description générale de ce que fait le module, avec un code d'exemple d'utilisation.
- Au début des fichiers d'implémentation, un entête spécifie :
  - les auteurs,
  - la licence, (copyright si rien)
  - une liste datée des modifications

commentaires

#QDLE#Q#A*BC#30#

```
int my_function(int arg){
     // set a counter to 0
     int counter=0 ;
     … ..
     … ..
     if (x==0){
          // because the string is empty in
          //this case
     }
     …
     …
     for(i=0;i<counter-1;++i){
          // parse the char of the
          // string avoiding the \0
          … .
     }
     … .
```

- sur cet exemple, quel commentaire est sans intérêt ?
  - A
  - B
  - C

Automatisation

- La compilation manuelle n'est pas possible sur un grand projet :
  - homogénéisation des options de compilation
  - gestion des dépendances (que doit-on recompiler?)
  - commandes et options spécifiques à une plate-forme.
  - erreurs manuelles
  - ...
- On doit se munir d'outils pour automatiser cette étape.

Makefile

- make est un outil qui permet d'automatiser la création et la mise à jour de fichiers.
- make repose sur un fichier de description : Makefile
- Un Makefile est un ensemble de règles formatées :

  cible : source1 source2 source3 ...

  commande1

  commande2

  ... .
- Si « cible » n'existe pas où est moins récente que l'une des sources, alors les commandes sont exécutées.

Makefile : exemple

```
image_thumbail.jpg : image.jpg
    convert -size 80x80 image.jpg image_thumbnail.jpg
image.jpg : image.png
    convert image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

- make est récursif : si un fichier source n'existe pas, il cherche une règle pour le créer.
- initialement, make cherche à créer une seule cible : la première du Makefile ou celle(s) indiquée(s) sur la ligne d'appel : make image.png
- Si une source est manquante et qu'il n'y a pas de règle pour la créer : erreur
- Si une des commandes renvoie une valeur différente de 0, make s'arrête (tips : `commande || true`)

Makefile : variables

- make supporte la déclaration de variable :
- `NOM=VALEUR` `(La valeur peut comporter des espaces)`
- $(NOM) est remplacé par VALEUR.
- Déclaration particulière :
  - `NOM+=VALEUR` (concaténation)
  - `NOM?=VALEUR` positionne la variable si elle n'existe pas
  - `NOM:= $(shell ls *.c)`
- Les variables d'environnement sont reprises par make :

```
$>    CFLAGS='-g' make
```

- On peut également fixer la valeur depuis les arguments :

```
$>    make CFLAGS='-g'
```

Makefile : variable, exemple

```
CONVERT=convert
THUMB_SIZE ?=80x80
image_thumbail.jpg : image.jpg
    $(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
image.jpg : image.png
    $(CONVERT) image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

$> THUMB_SIZE=60x60 make

Makefile : spéciales

- Quelques variables spéciales pour l'écriture des commandes :

```
cible : source1 source2
```

  - $@ : cible
  - $< : source1
  - $^ : source1 source2

- Ainsi :

```
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
```

- Devient :

```
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) $< $@
```

Makefile : règles génériques

- Il est possible d'écrire des règles génériques :

```
%.jpg : %.png
	convert $< $@
```

- Permet de convertir tout fichier png en jpg à l'aide du programme *convert*

```
%_thumb.jpg : %.jpg
	convert -size 80x80 $< $@
```

- Il est enfin possible de séparer la liste des dépendances et la règle de création.

Makefile et compilation

```
CC=gcc
CFLAGS-=-Wall

prog : example.o hash.o
    $(CC) -o $@ $^
exemple.o : exemple.c hash.h
hash.o : hash.c hash.h
%.o :
    $(CC) $(CFLAGS) $< -o $@
```

- La dernière règle explique comment générer un fichier .o
- les deux règles au dessus donnent les dépendances

Dépendances

- C’est au développeur d’indiquer les dépendances.
- Ainsi, si un fichier header est modifié, tout fichier source qui inclut directement ou indirectement ce header doit être recompilé
- On peut demander à gcc d’analyser les fichiers et produire les règles de dépendances:

```
$> gcc -MM hash.c exemple.c
hash.o: hash.c hash.h
exemple.o: exemple.c hash.h
```

- Le résultat peut être inclus dans le Makefile avec la fonction *include*

include et gcc -MM

```
CC=gcc
CFLAGS=-Wall

prog: hash.o exemple.o
	$(CC) $^ -o $@

include dep
dep: hash.c exemple.c hash.h
	gcc -MM $^ > dep
```

```
pg106$ make
make: 'prog' is up to date.
pg106$ touch exemple.c
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o exemple.o exemple.c
gcc hash.o exemple.o -o prog
pg106$ touch hash.h
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o hash.o hash.c
gcc -Wall   -c -o exemple.o exemple.c
gcc hash.o exemple.o -o prog
pg106$ touch hash.c
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o hash.o hash.c
gcc hash.o exemple.o -o prog
pg106$
```

Makefile : PHONY

- On peut vouloir écrire des règles qui ne génèrent pas de fichier :

```
all

install

clean

distclean
```

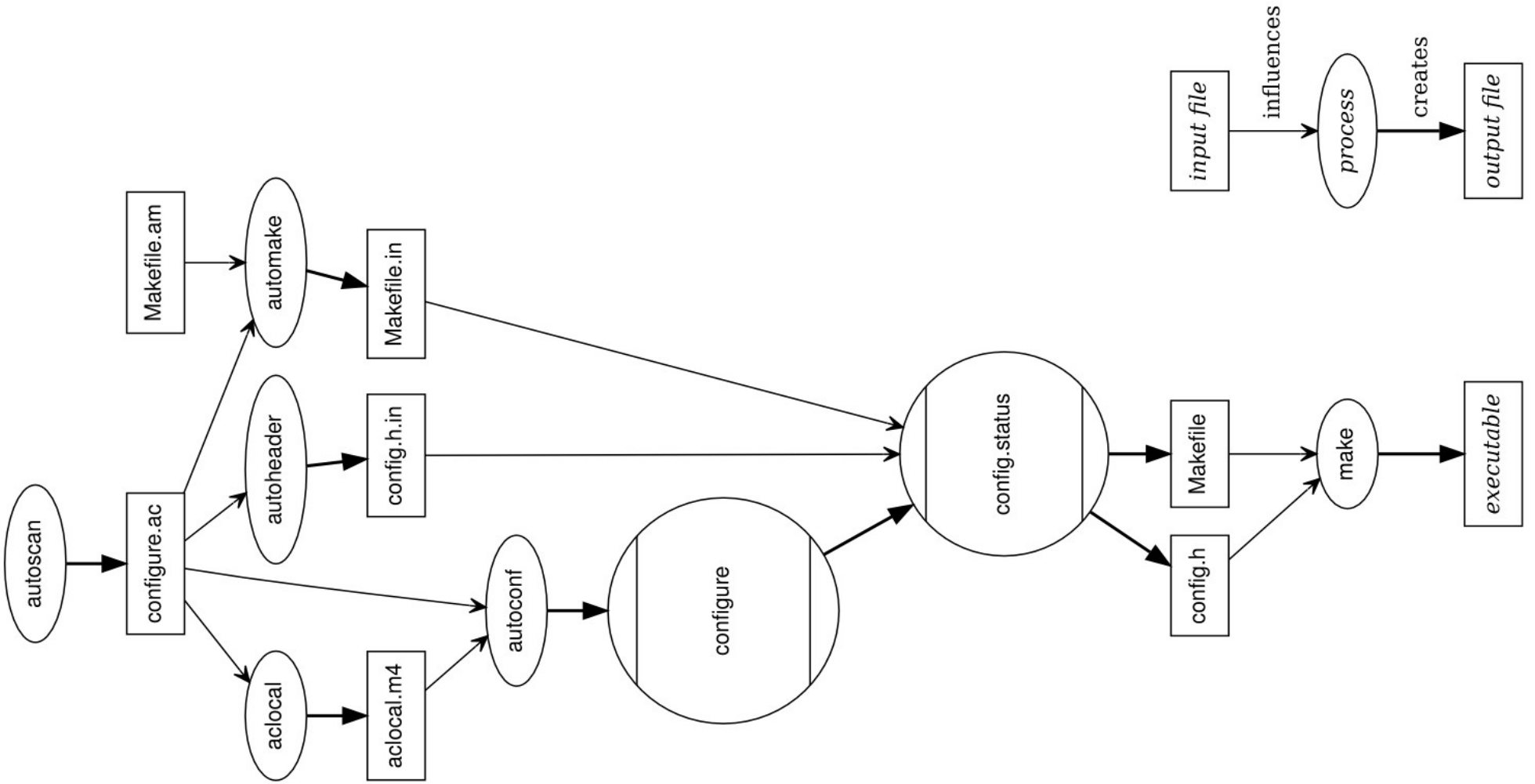
- On indique cela à make :

```
.PHONY=all install clean distclean
```

Makefile, automake...

- Makefile n'est pas spécifique à la compilation de programme
- Il ne gère pas nativement la dépendance entre les fichiers sources
- La prise en compte de l'environnement (compilateur, options, bibliothèques…) peut ce faire :
  - par l'édition des variables du Makefile (options de compilations, répertoire d'installation...)
  - par l’écriture de plusieurs Makefile (un par système)
  - par l'utilisation d'un générateur :
    - automake / autoconf
    - cmake

auto-tools : automake / autoconf



Auto-tools : principe général

- Le développeur écrit un fichier *configure.ac* :

```
AC_INIT([penguin], [2019.3.6], [seth@example.com])
AC_OUTPUT
AM_INIT_AUTOMAKE
AC_CONFIG_FILES([Makefile])
AC_PROG_CXX
```

- Le programme *autoconf* va utiliser les informations pour générer un script *configure*
- Le développeur écrit également un fichier *Makefile.am* :

```
bin_PROGRAMS = penguin
penguin_SOURCES = penguin.cpp
```

- Le script configure va utiliser ce fichier pour produire un **Makefile**

cmake

cmake

- CMake est un outil simplifié permettant la compilation de sources C et C++.
- C'est un outil multi-plateformes sous licence BSD
- Nécessite la présence d'un fichier **CMakeLists.txt**

- gestion automatique des dépendances
- Simple d'utilisation / facile à prendre en main

cmake : hello world !

- Un seul fichier main.c que l'on souhaite compiler en un programme « hello_world » :

```
project(HelloWorld)
cmake_minimum_required(VERSION 3.0)

add_executable(hello_world main.c)
```

- CMakeList.txt peut être découpé sur plusieurs répertoires avec des inclusions
- Un projet <<HELLO>> avec une bibliothèque dans le répertoire Hello et un programme d'exemple dans le répertoire Demo

cmake

- ./CMakeLists.txt

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_library(hello hello.c)

add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo hello)
```

- Si je structure mon projet en deux répertoires: *hello* pour la bibliothèque et *demo* pour un exemple d’utilisation...

cmake

- ./CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_subdirectory (hello)
add_subdirectory (demo)
```

- ./hello/CMakeLists.txt

```
add_library (hello hello.c)
target_include_directories (hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

- ./demo/CMakeLists.txt

```
add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo LINK_PUBLIC hello)
```

cmake:cross platform make

- cmake est un système de compilation cross-plateformes. Il ne compile pas directement mais génère des fichiers dans différents formats :
  - Makefile
  - projet Visual Studio
  - Borland Makefile
  - projet Xcode
  - Kate
  - …
- cmake utilise les fichiers CMakeLists.txt et génère des fichiers en fonction de la plate-forme de compilation (Makefile, visual, xcode….).

cmake : les variables

- La déclaration de variables :
  - set(NAME VALUE)
  - ${NAME}
  - lors de l'appel à cmake : cmake -DNAME=VALUE
- Variables standards :
  - CMAKE_INCLUDE_PATH (pour les .h)
  - CMAKE_LIBRARY_PATH (pour la recherche de .so)
  - DESTDIR (pour l'installation)
  - CMAKE_BUILD_TYPE (Debug, Release)
- Dans le CMakeLists.txt :
  - CMAKE_C_FLAGS
  - CMAKE_C_FLAGS_DEBUG
  - CMAKE_C_FLAGS_RELEASE
  - CMAKE_CURRENT_SOURCE_DIR
  - CMAKE_CURRENT_BINARY_DIR
  - CMAKE_SOURCE_DIR

cmake : les fonctions

- add_executable(name sources)
- add_library(name STATIC sources)
- add_library(name SHARED sources)
- target_link_libraries(name libs)
- include_directories(dir1 dir2…)
- add_custom_command

cmake : utilisation

- cmake support l'out-source building : c'est à dire la compilation **en dehors** du répertoire des sources
- On suppose : projet/CMakeLists.txt
- alors on peut faire :

```
mkdir projet-build ; cd projet-build
cmake ../projet
make
```

- et

```
mkdir projet-debug ; cd projet-debug
cmake -DCMAKE_BUILD_TYPE=Debug ../projet
make
```

CMake: les modules

- Un module est un fichier écrit dans le langage de cmake avec l'extension: « .cmake »
- Les modules Find*Module*.cmake permettent de vérifier qu'un module est bien accessible et de positionner au besoin des variables.
- Par exemple « FindZLIB.cmake » vérifie que la bibliothèque libzlib.so et les headers sont bien disponibles sur le système. Des variables comme ZLIB_FOUND, ZLIB_LIBRARIES, ZLIB_INCLUDE_DIRS sont positionnées.

CMake: utilisation d’un module

- Dans un projet on écrit:

  find_package(ZLIB Required)

  include_directories(${ZLIB_INCLUDE_DIRS})

  ...

  target_link_libraries(hello ${ZLIB_LIBRARIES})

- Si zlib n’est pas trouvée, alors cmake s’arrêtera avec un message d’erreur lors de la phase de configuration.
- La commande *cmake --help-module-list* permet de lister l’ensemble des modules disponibles.